



# 嵌入式系统

## Embedded System

毛维杰

杭州 • 浙江大学 • 2021

# 第4章 汇编语言与**C51**程序设计

- § 4-1 程序设计概述
- § 4-2 汇编语言程序设计
- § 4-3 汇编语言程序设计实例
- § 4-4 **C51**特点及其程序结构
- § 4-5 **C51**程序设计实例

# § 4-1 程序设计概述

## 程序语言分类

- 1. 机器语言：机器语言是用二进制代码0和1表示指令和数据的最原始的程序设计语言。
- 2. 汇编语言：在汇编语言中，指令用助记符表示，地址、操作数可用标号、符号地址及字符等形式来描述。
- 3. 高级语言：高级语言是接近于人的自然语言，面向过程而独立于机器的通用语言。

## § 4-2 汇编语言程序设计

### 伪操作指令

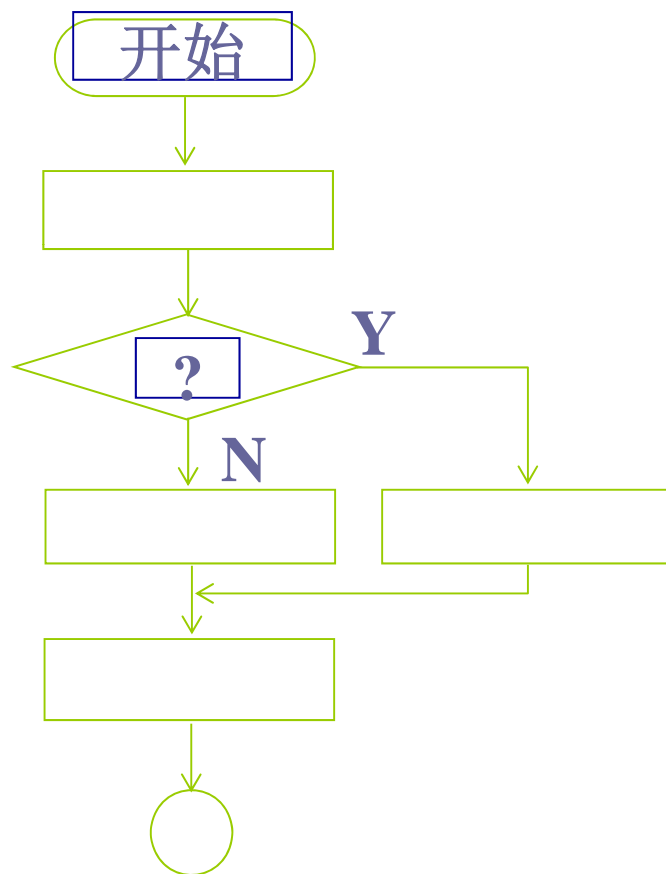
- 1、 **ORG** (**Origin**) 定义程序的起始地址
- 2、 **END** 程序结束标志
- 3、 **EQU** (**Equate**) 表达式赋值
- 4、 **DB** (**Define Byte**) 定义字节
- 5、 **DW** (**Define Word**) 定义字
- 6、 **BIT** 位地址赋值(绝对地址或符号地址)

# 汇编语言格式

地址	机器码	源程序	注释
		<b>ORG 0000H</b>	； 整个程序起始地址
<b>0000</b>	<b>20 00 30</b>	<b>LJMP MAIN</b>	； 跳向主程序
		<b>ORG 0030H</b>	； 主程序起始地址
<b>0030</b>	<b>C3</b>	<b>MAIN: CLR C</b>	； <b>MAIN</b> 为程序标号
<b>0031</b>	<b>E6</b>	<b>LOOP: MOV A, @R0</b>	
<b>0032</b>	<b>37</b>	<b>ADDC A, @R1</b>	
<b>0033</b>	<b>08</b>	<b>INC R0</b>	
<b>0034</b>	<b>DA FB</b>	<b>DJNZ R1, LOOP</b>	； 相对转移
<b>0036</b>	<b>80 03</b>	<b>SJMP NEXT</b>	
<b>0038</b>	<b>78 03</b>	<b>MOV R0, #03H</b>	
<b>003A</b>	<b>18</b>	<b>NEXT: DEC R0</b>	
<b>003B</b>	<b>80FE</b>	<b>SJMP \$</b>	； <b>HERE: SJMP HERE</b>
		<b>END</b>	； 结束标记

# 汇编语言程序设计步骤

1. 确定方案和计算方法
2. 了解应用系统的硬件配置、性能指标。
3. 建立系统数学模型，确定控制算法和操作步骤。
4. 画程序流程图，确定程序的流向
5. 编制源程序
  - (1) 合理分配存储器单元和了解I/O接口地址。
  - (2) 按功能设计程序，明确各程序之间的相互关系。
  - (3) 用注释行说明程序，便于阅读和修改调试和修改。



## § 4-3 汇编语言程序设计实例

### ■ 顺序程序结构

例：两个无符号双字节数相加（子函数）。

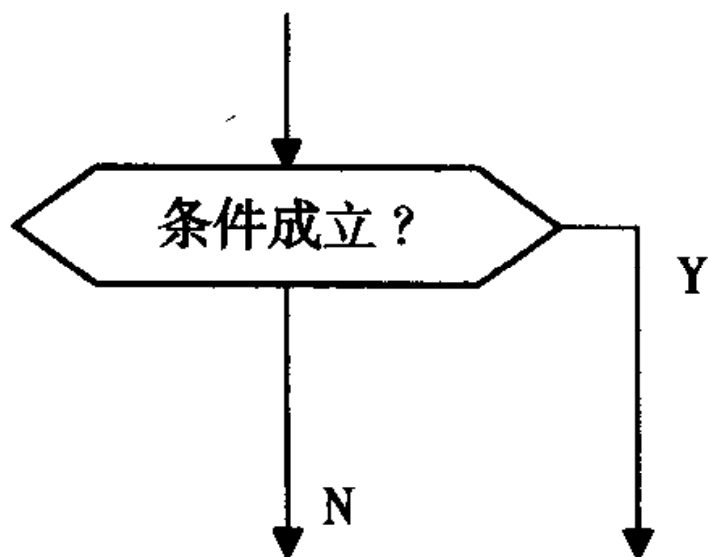
- 设被加数存放于内部RAM的40H（高位字节），41H（低位字节），加数存放于50H（高位字节），51H（低位字节），和数存入 40H和41H单元中。
- R0、R1中存放数据地址。

程序如下：

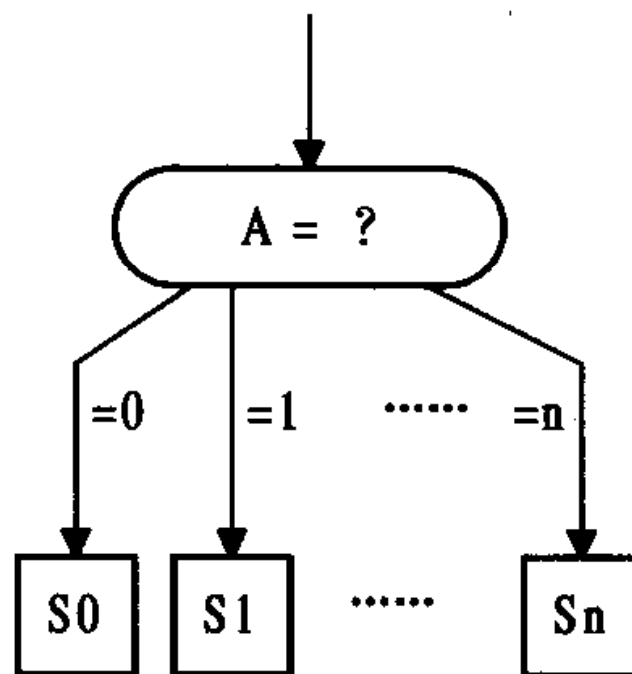
START: CLR C	; 将Cy清零
AD1: MOV A, @R0	; 被加数低字节的内容送入A
ADD A, @R1	; 两个低字节相加
MOV @R0, A	; 低字节的和存入被加数低字节中
DEC R0	; 指向被加数高位字节
DEC R1	; 指向加数高位字节
MOV A, @R0	; 被加数高位字节送入A
ADDC A, @R1	; 两个高位字节带Cy相加
MOV @R0, A	; 高位字节的和送被加数高位字节
RET	



## ■ 分支程序设计



(a)



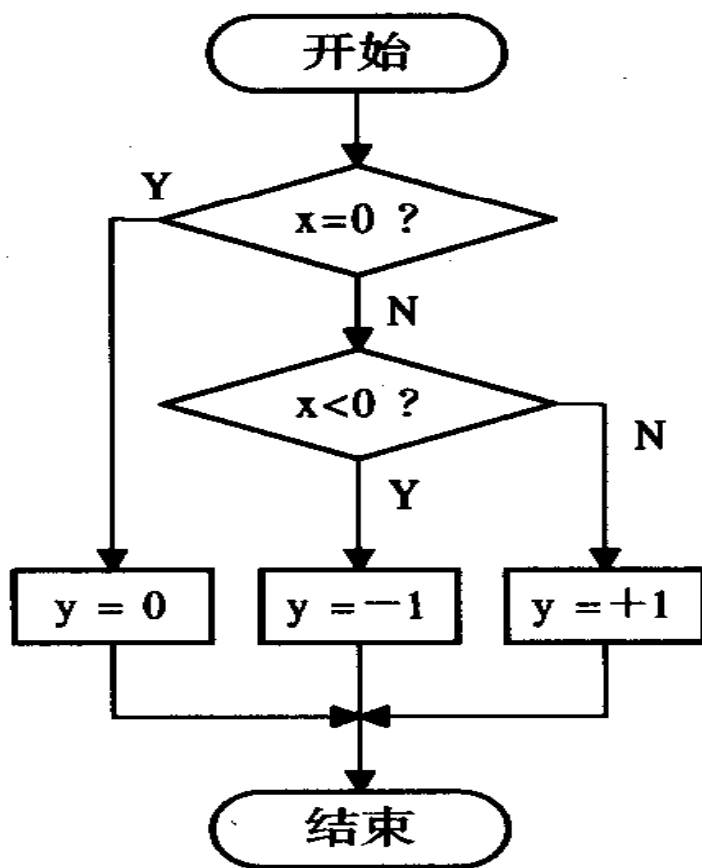
(b)

分支结构框图

(a) 单分支流程; (b) 多分支流程

例：x, y均为8位二进制数, 设 x存入R0, y存入R1, 求解:

$$y = \begin{cases} +1 & x > 0 \\ -1 & x < 0 \\ 0 & x = 0 \end{cases}$$



程序如下:

START: CJNE R0, # 00H, SUL1 ; R0中的数与00比较不等转移

MOV R1, # 00H ; 相等,  $R1 \leftarrow 0$

SJMP SUL2

SUL1: JC NEG ; 两数不等, 若  $(R0) < 0$ , 转向NEG

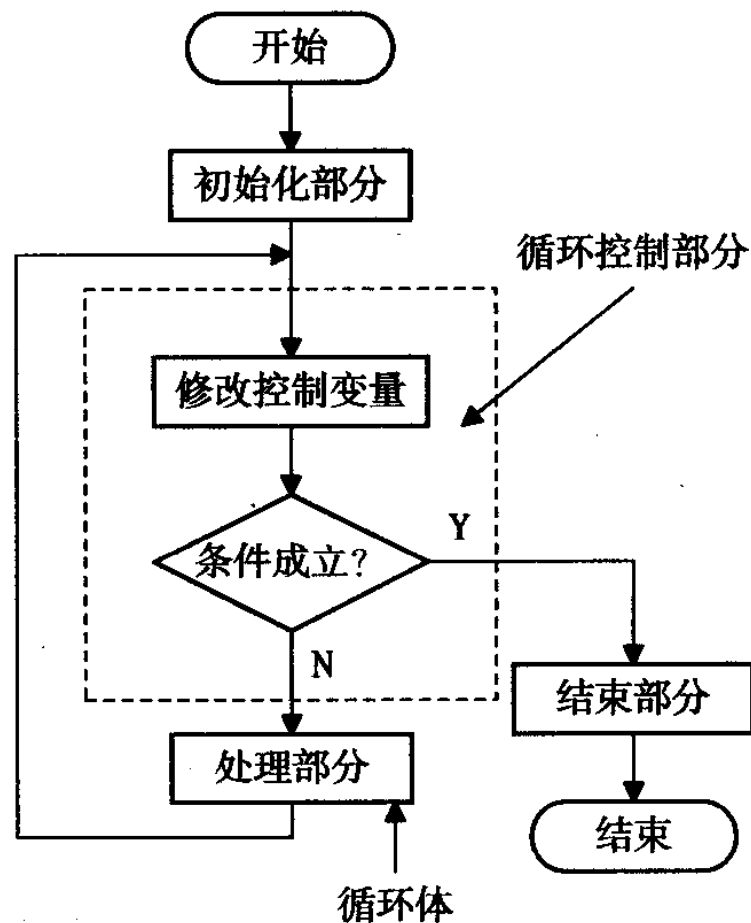
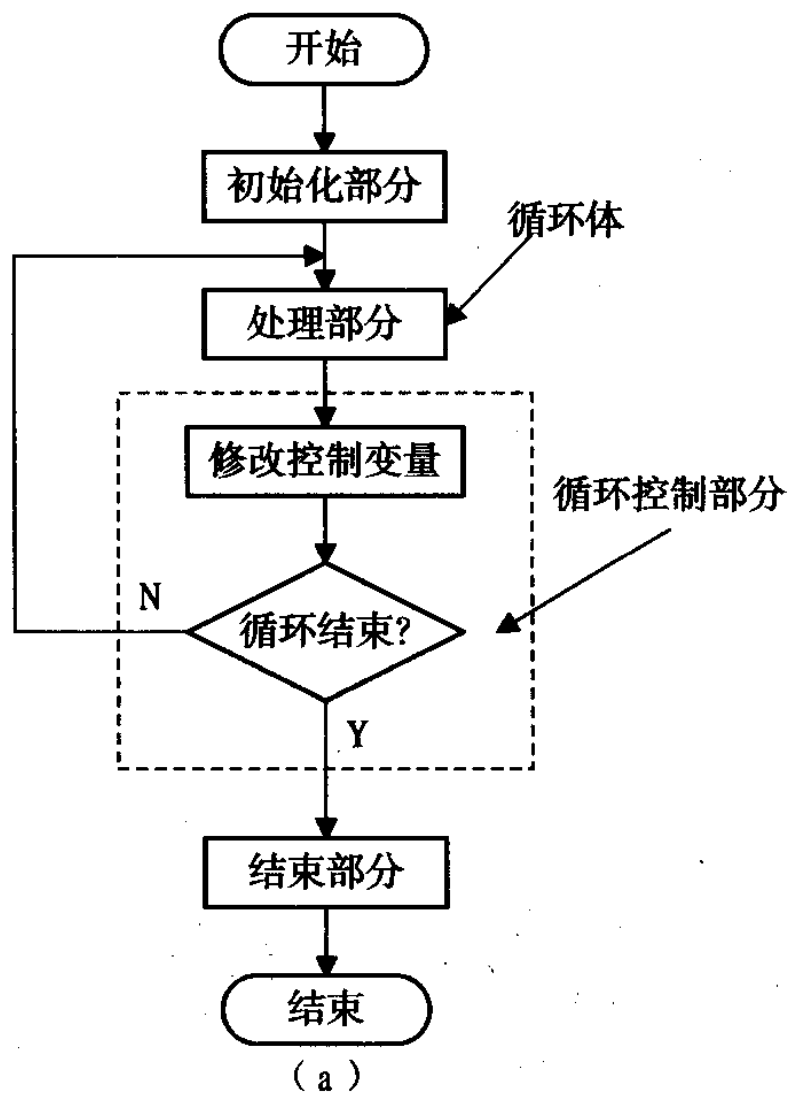
MOV R1, # 01H ;  $(R0) > 0$ , 则  $R1 \leftarrow 01H$


SJMP SUL2

NEG: MOV R1, # 0FFH ;  $(R0) < 0$ , 则  $R1 \leftarrow 0FFH$

SUL2: RET

# ■ 循环程序设计





例： 工作单元清零。

在应用系统程序设计时，有时经常需要将存储器中部分地址单元作为工作单元，存放程序执行的中间值或执行结果，工作单元清零工作常常放在程序的初始化部分中。

设有50个工作单元，其首址为外部存储器8000H单元，则其工作单元清零程序如下：



CLEAR: CLR A

MOV DPTR, # 8000H ; 工作单元首址送指针

MOV R2, #50 ; 置循环次数

CLEAR1: MOVX @DPTR, A

INC DPTR ; 修改指针

DJNZ R2, CLEAR1 ; 控制循环

RET

例：设在内部RAM的BLOCK单元开始处有长度为LEN的无符号数据块，试编一个求和程序，并将和存入内部RAM的SUM单元（设和不超过8位）。

```
BLOCK EQU 20H
LEN     EQU 60H
SUM     EQU 61H

START:  CLR A                ;清累加器A
        MOV R2, #LEN         ;数据块长度送R2
        MOV R1, #BLOCK      ;数据块首址送R1
LOOP:   ADD A, @R1           ;循环加法
        INC R1               ;修改地址指针
        DJNZ R2, LOOP        ;修改计数器并判断
        MOV SUM, A           ;存和
        RET
```


# 多重循环

例： 10 秒延时程序。

延时程序与 MCS - 51 执行指令的时间有关, 如果使用 12 MHz 晶振, 一个机器周期为  $1\ \mu\text{s}$ , 计算出一条指令以至一个循环所需要的执行时间, 给出相应的循环次数, 便能达到延时的目的。10 秒延时程序如下:

```
DELAY: MOV R5, # 100
      DEL0: MOV R6, # 200
      DEL1: MOV R7, # 248          ; 1  $\mu\text{s}$ 
            NOP                    ; 1  $\mu\text{s}$ 
      DEL2: DJNZ R7, DEL2           ;  $2\ \mu\text{s} \times 248 = 496\ \mu\text{s}$ 
            DJNZ R6, DEL1           ;  $(496 + 1 + 1 + 2)\ \mu\text{s} \times 200 = 0.1\text{s}$ 
            DJNZ R5, DEL0           ;  $(100000 + 1 + 2)\ \mu\text{s} \times 100 = 10.0003\text{s}$ 
            RET                     ;  $10.0003\text{s} + (2 + 1)\ \mu\text{s}$ 
```



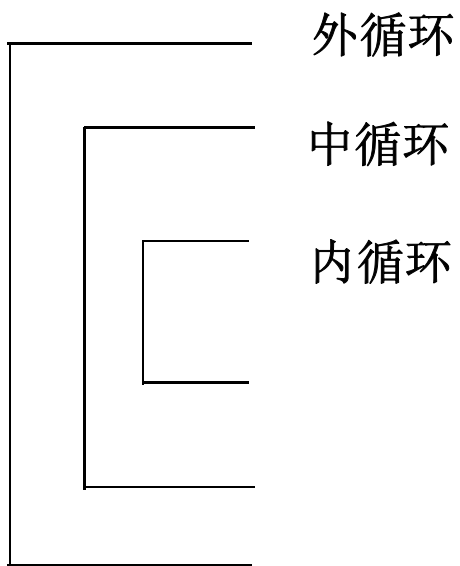


上例程序中采用了多重循环程序，即在一个循环体中又包含了其它的循环程序，这种方式是实现延时程序的常用方法。使用多重循环时，必须注意：

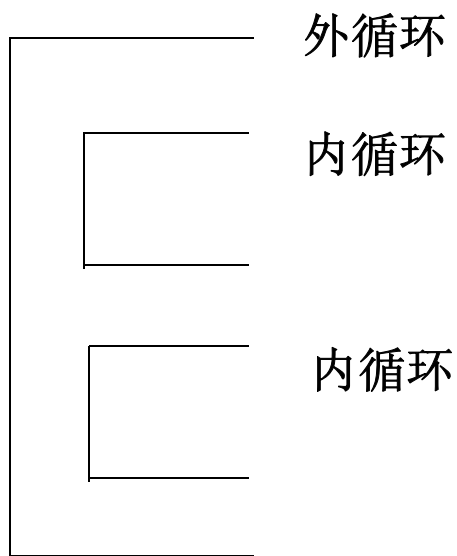
(1) 循环嵌套，必须层次分明，不允许产生内外层循环交叉。

(2) 外循环可以一层层向内循环进入，结束时由里往外一层层退出。

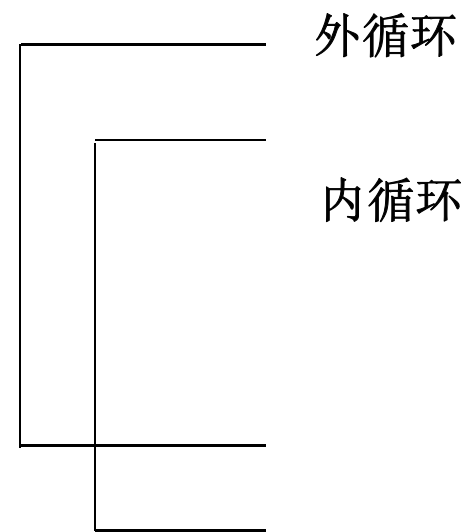
(3) 内循环可以直接转入外循环，实现一个循环由多个条件控制的循环结构方式。



(a) 嵌套正确



(b) 嵌套正确



(c) 交叉不正确

多重循环示意图

例：在内部 RAM 中从 50H 单元开始的连续单元依次存放了一串字符，该字符串以回车符为结束标志，要求测试该字符串的长度。

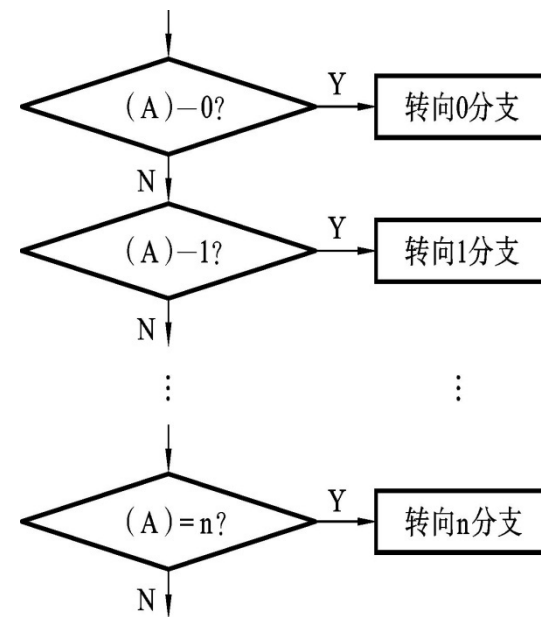
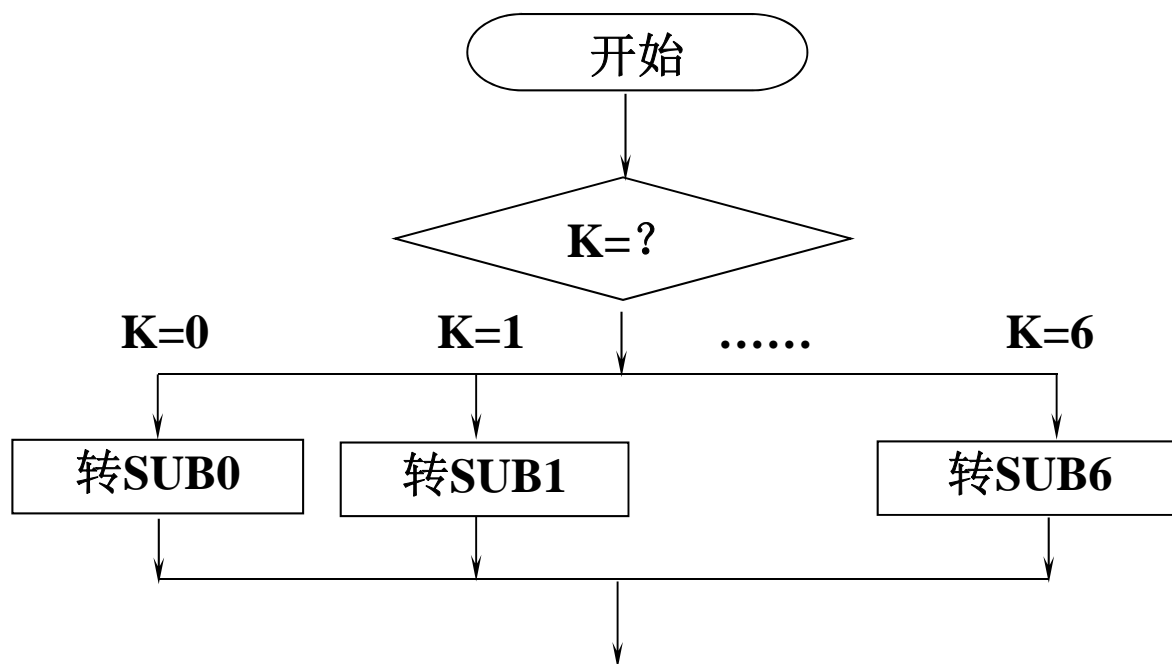
程序如下：

```
START:  MOV R2, #0FFH
        MOV R0, #4FH           ; 数据指针R0置初值
LOOP:   INC R0
        INC R2
        CJNE @R0, #0DH, LOOP
        RET
```

## ■ 散转程序设计

1) 多次使用条件转移, 例如 **CJNE A, #data, rel**, 转向不同的分支入口

2) 使用 **JMP @A+DPTR** 指令,  
转向不同的分支入口



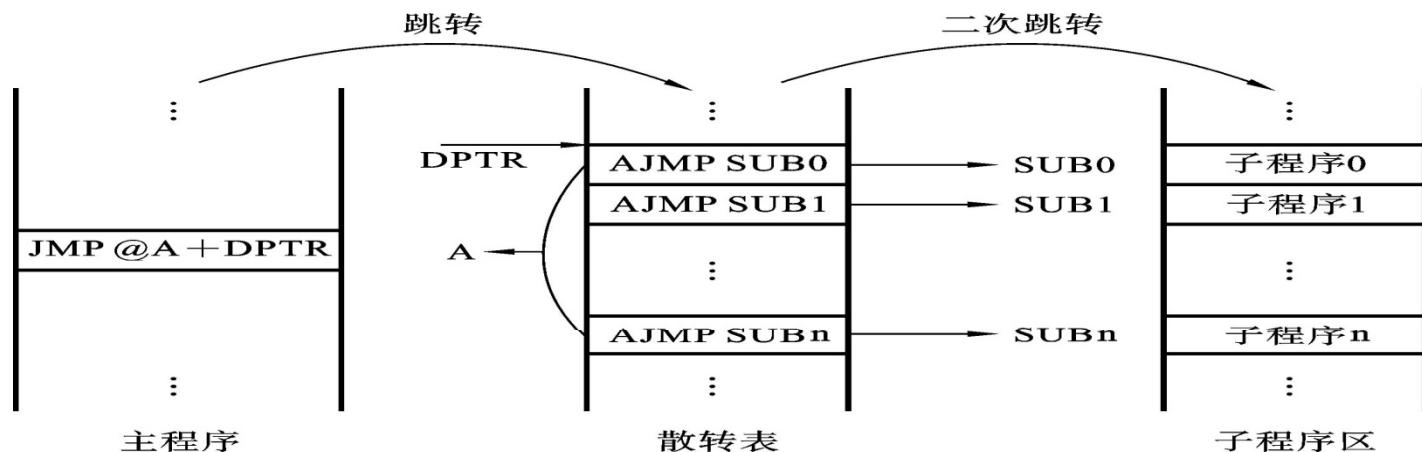
# 使用JMP @A+DPTR指令，转向不同的分支入口

利用这条指令实现程序散转的思路有两个：

- ①是数据指针固定，根据累加器的内容，程序转入相应的分支程序中。
- ②是累加器清零，根据数据指针的值，决定程序转向的目的地址。

## (1) 采用转移指令表的散转程序

- ①构建一个由到相应子程序段的转移指令组成的转移指令表（用AJMP或LJMP指令组成）。
- ②用JMP @A+DPTR指令使程序跳转到转移指令表相应的位置，从而实现到相应子程序段的跳转（如图）。



## 例：根据R2的内容，转向各个处理程序

(R2) = 0, 转向 PRG0

(R2) = 1, 转向 PRG1

...

(R2) = n, 转向 PRGn

### 参考程序：

```
JMP1:  MOV DPTR, # TBJ1; 指向跳转表首址
        MOV  A, R2
        ADD  A, R2  ; 因为AJMP指令为两个字节
        JNC  NADD   ; R2*2<256时，直接到NADD散转
        INC  DPH     ; R2*2>256时，表空间增加一页，修改DPH
NADD:   JMP   @A+DPTR
TBJ1:   AJMP PRG0    ; 建立中间散转表。AJMP指令占两个字节。
        AJMP PRG1    ; AJMP指令的寻址范围为2K字节，若n个散转分支
        ...         ; 程序长度超过2K，可使用LJMP（三字节指令）
        ...         ;
        AJMP PRGn
PRG0:   ...         ; 各处理程序，地址间隔无规律
        ...
PRG1:   ...         ...
        ...         ...
PRGn:   ...         ...
```

## (2) 用地址偏移表实现散转

上面介绍的方法是用两次转移来实现散转的，第一次由JMP指令进入转移表，再由转移指令AJMP或LJMP指令转入2K或64K的存储空间。如果散转点较少且所有处理程序在同一页(256字节)时，就可以直接在表中给出散转程序的入口偏移地址。

例：根据R7的内容(0, 1, 2, 3)实现散转。

该程序的表格中存放的各分程序的入口偏移地址，并在基地址TAB的基础上实现散转。程序如下：

```
ORG 2000H
MOV A, R7
MOV DPTR, #TAB
MOVC A, @A+DPTR    ; 取出分程序的偏移地址
JMP  @A+DPTR        ; 转分程序
TAB:  DB  rel0       ; 分程序0的偏移地址
      DB  rel1       ; 分程序1的偏移地址
      DB  rel2       ; 分程序2的偏移地址
      DB  rel3       ; 分程序3的偏移地址
```

假设R7=0，rel0=30H，则执行上面程序后，将转移到地址为TAB+30H处执行分程序0。

### (3) 用转向地址表实现散转

一种方法是将A清零，根据DPTR的值来决定程序转向的目标地址。DPTR的内容可以通过查表或其它方法获得。

例：根据R7的内容转入各相应的操作程序中。

```
JMUP4: MOV    DPTR, #TAB4
        MOV    A, R7
        ADD    A, R7    ; R7*2
        JNC    NADD
        INC    DPH    ; R7*2进位到DPH
NADD:   MOV    R3, A    ; 暂存
        MOVC   A, @A+DPTR
        XCH    A, R3
        INC    A
        MOVC   A, @A+DPTR
        MOV    DPL, A
        MOV    DPH, R3
        CLR    A
        JMP    @A+DPTR
TAB4:   DW     OPR0
        DW     OPR1
        .....
        DW     OPRn
```



## 综合应用举例：数据排序程序

数据的排序，其算法很多，常用的有插入排序法、冒泡排序法、快速排序法、选择排序法、堆积排序法、二路归并排序法及基数排序法等。

冒泡法是一种相邻数互换的排序方法，其过程类似水中气泡上浮。执行时从前向后进行相邻数比较，若数据的大小次序与要求的顺序不符时(逆序)，就将二数互换，正序时不交换，假定是升序排序，则通过这种相邻数互换方法，使小数向前移，大数向后移，为此从前向后进行一次冒泡（相邻数互换），就会把最大数换到最后，再进行一次冒泡，就会把次大数排在倒数第二，直至冒泡结束。


## 说明:

(1) 每次冒泡都从前向后排定了一个大数（升序），每次冒泡所需进行的比较次数都递减，例如有 $n$ 个数排序，则第一次冒泡需比较（ $n-1$ ）次，第二次冒泡则需（ $n-2$ ）次，依此类推。

(2) 对于 $n$ 个数，理论上说应进行（ $n-1$ ）次冒泡才能完成排序，但实际上往往不到（ $n-1$ ）次就已排好序。判定排序是否完成的最简单方法是每次冒泡中是否有互换发生，如果有互换发生，说明排序还没完成，否则就表示已排好序，为此控制排序结束常不使用计数方法，而使用设置互换标志的方法，以其状态表示在一次冒泡中是否有数据互换进行。

# 冒泡排序

- 有N个无符号数依次存放于内部**RAM**以**BLOCK**为首地址的区域中，编程将它们从小到大排列，并存放在原区域中。



```
BUBBLE:      MOV R1, #N-1
              MOV R0, #BLOCK
              CLR F0                ;Clear flag
BUBLOOP:     MOV A, @R0
              INC R0
              MOV 20H, @R0
              CJNE A, 20H, BUBLOOP1
BUBLOOP1:    JC BUBNEXT            ;A<=(20H), NO EXCHANGE
              DEC R0
              MOV @R0, 20H
              INC R0
              MOV @R0,A
              SETB F0
BUBNEXT:     DJNZ R1, BUBLOOP
              JB F0, BUBBLE
              SJMP $
              END
```



## 子程序和参数传递

通常把一些基本操作功能编制为程序段作为独立的子程序，以供不同程序或同一程序反复调用。在程序中需要执行这种操作的地方放置一条调用指令，当程序执行到调用指令，就转到子程序中完成规定的操作，并返回到原来的程序继续执行下去。

## 参数传递一般可采用以下方法:

- 传递数据。将数据通过工作寄存器R0~R7或累加器来传送。即主程序和子程序在交接处,上述寄存器和累加器存储的是同一参数。
- 传送地址。数据存放在数据存储器中,参数传递时只通过R0、R1、DPTR传递数据所存放的地址。
- 通过堆栈传递参数。在调用之前,先把要传送的参数压入堆栈,进入子程序之后,再将压入堆栈的参数弹出到工作寄存器或者其他内存单元。

## § 4-4 C51语言特点及其程序结构

### C51语言的特点:

- 1. 语言简洁、紧凑，使用方便、灵活；
- 2. 运算符极其丰富；
- 3. 生成的目标代码质量高，程序执行效率高  
(比汇编语言编写程序低10%—20%) ；
- 4. 可移植性好；
- 5. 可以直接操作硬件。

# C-51与ASM-51相比有如下优点:

1. 对指令系统不必深入了解，只要对存贮器空间结构有深入了解；
2. 寄存器分配、不同存贮器的寻址及数据类型等细节可由编译器管理；
3. 程序有规范的结构，可分成不同的函数，这种方式可使程序结构化；
4. 具有将可变的选择不与特殊操作组合在一起的能力，改善了程序的可读性；
5. 提供的库包含许多标准子程序，具有较强的数据处理能力；
6. 具有方便的模块化编程技术，使程序很容易移植。



# C51的基本数据类型

类型	符号	关键字	所占 位数	数的表示范围
整 型	有	(signed) int	16	-32768 ~ 32767
	有	(signed) long int	32	-2147483648 ~ 2147483647
	无	unsigned int	16	0 ~ 65535
	无	unsigned long int	32	0 ~ 4294967295
实型	有	float	32	-3.4e38 ~ 3.4e38
字符 型	有	char	8	-128 ~ 127
	无	unsigned char	8	0 ~ 255
位型	无	bit	1	0或1

# C51数据的存储类型

名称	存储空间位置	位数	范围	说 明
data	直接寻址片内 <b>RAM</b>	8	0~127	片内RAM 00~7FH的128个字节, 访问速度最快
bdata	可位寻址片内RAM	1	0/1	位寻址片内RAM20~2FH, 位与字节可混合访问
idata	间接寻址片内RAM	8	0~255	00~FFH的128个片内RAM, 及52子系列的高128字节内部RAM。访问方式: MOV A, @Ri
pdata	片外页RAM	8	0~255	寻址片外RAM 低256字节, 由MOVX A, @Ri访问
xdata	片外RAM	16	0~65535	片外RAM全部64KB, 由MOVX A, @DPTR访问
code	程序ROM	16	0~65535	ROM区全部64KB, 由MOVC A, @A+DPTR访问

## 两个关键字: sfr和sbit

(标准SFR在reg51.h、reg52.h 等头文件中已经被定义, 只要用文件包含做出申明即可使用)

1. 定义特殊功能寄存器用sfr
2. 定义可位寻址对象, 如访问特殊功能寄存器中的某位用sbit

# C51数据的存储模式（编译模式）

存储模式	说 明
SMALL	默认data，参数及局部变量放入可直接寻址片内RAM的用户区中(最大128字节)。另外所有对象(包括堆栈)，都必须嵌入片内RAM。
COMPACT	默认pdata，参数及局部变量放入分页的外部数据存储区，通过@R0或@R1间接访问，栈空间位于片内数据存储区中。
LARGE	默认xdata，参数及局部变量直接放入片外数据存储区，使用数据指针DPTR来进行寻址。用此数据指针进行访问效率较低，尤其对两个或多个字节的变量，这种数据类型的访问机制直接影响代码的长度。

建议首先选用SMALL

# C51的常量、运算符、表达式、基本语句、数组、函数

一、C51的常量

与标准C相同

二、C51常用运算符

与标准C基本一致

三、C51表达式

与标准C基本一致

四、C51的基本语句

与标准C基本一致

五、C51的数组

与标准C基本一致

六、C51函数的定义

与标准C基本一致

# C51指针

## 1. 存储器指针

基于存储器的指针是在说明一个指针时，指定它所指向的对象的存储类型。长度为1或2字节，例如：

```
char data *px;
```

```
char xdata *py;
```

## 2. 一般指针

不作特别声明的指针，即为一般指针。例如： `char *pz;`

这里没有指定指针变量所指向的变量的存储类型，处于编译模式默认的存储区，长度为3字节。3个字节的含义如下：

地址	+0	+1	+2
内容	存储类型的编码	高位地址偏移量	低位地址偏移量

存储类型	idata	xdata	pdata	data	code
编码值	1	2	3	4	5

# C51程序结构

包含<头文件>

函数类型说明

全程变量定义

main()

{

    局部变量定义

    <程序体>

}

func1()

{

    局部变量定义

    <程序体>

}

... ..

funcN()

{

    局部变量定义

    <程序体>

}

## 常用C51的头文件:

reg51.h （定义特殊功能寄存器等）；

math.h （数学函数）；

ctype.h （字符函数）；

stdio.h （一般IO函数）；

stdlib.h （标准函数）；

absacc.h （绝对地址访问）；

string.h （串函数）

... ..

# 常用C51的头文件

```
/*-----
```

REG51.H

Header file for generic 80C51 and 80C31 microcontroller.

Copyright (c) 1988-2002 Keil Elektronik GmbH and Keil Software, Inc.

All rights reserved.

```
-----*/
```

```
#ifndef __REG51_H__
```

```
#define __REG51_H__
```

```
/* BYTE Register */
```

```
sfr P0 = 0x80;
```

```
sfr P1 = 0x90;
```

```
sfr P2 = 0xA0;
```

```
sfr P3 = 0xB0;
```

```
sfr PSW = 0xD0;
```

```
sfr ACC = 0xE0;
```

```
sfr B = 0xF0;
```

```
sfr SP = 0x81;
```

```
sfr DPL = 0x82;
```

```
sfr DPH = 0x83;
```

```
sfr PCON = 0x87;
```

```
sfr TCON = 0x88;
```

```
sfr TMOD = 0x89;
```

```
sfr TL0 = 0x8A;
```

```
sfr TL1 = 0x8B;
```

```
sfr TH0 = 0x8C;
```

```
sfr TH1 = 0x8D;
```

```
sfr IE = 0xA8;
```

```
sfr IP = 0xB8;
```

```
sfr SCON = 0x98;
```

```
sfr SBUF = 0x99;
```

```
/* BIT Register */
```

```
/* PSW */
```

```
sbit CY = 0xD7;
```

```
sbit AC = 0xD6;
```

```
sbit F0 = 0xD5;
```

```
sbit RS1 = 0xD4;
```

```
sbit RS0 = 0xD3;
```

```
sbit OV = 0xD2;
```

```
sbit P = 0xD0;
```

```
/* TCON */
```

```
sbit TF1 = 0x8F;
```

```
sbit TR1 = 0x8E;
```

```
sbit TF0 = 0x8D;
```

```
sbit TR0 = 0x8C;
```

```
sbit IE1 = 0x8B;
```

```
sbit IT1 = 0x8A;
```

```
sbit IE0 = 0x89;
```

```
sbit IT0 = 0x88;
```

```
/* IE */
```

```
sbit EA = 0xAF;
```

```
sbit ES = 0xAC;
```

```
sbit ET1 = 0xAB;
```

```
sbit EX1 = 0xAA;
```

```
sbit ET0 = 0xA9;
```

```
sbit EX0 = 0xA8;
```

```
/* IP */
```

```
sbit PS = 0xBC;
```

```
sbit PT1 = 0xBB;
```

```
sbit PX1 = 0xBA;
```

```
sbit PT0 = 0xB9;
```

```
sbit PX0 = 0xB8;
```

```
/* P3 */
```

```
sbit RD = 0xB7;
```

```
sbit WR = 0xB6;
```

```
sbit T1 = 0xB5;
```

```
sbit T0 = 0xB4;
```

```
sbit INT1 = 0xB3;
```

```
sbit INT0 = 0xB2;
```

```
sbit TXD = 0xB1;
```

```
sbit RXD = 0xB0;
```

```
/* SCON */
```

```
sbit SM0 = 0x9F;
```

```
sbit SM1 = 0x9E;
```

```
sbit SM2 = 0x9D;
```

```
sbit REN = 0x9C;
```

```
sbit TB8 = 0x9B;
```

```
sbit RB8 = 0x9A;
```

```
sbit TI = 0x99;
```

```
sbit RI = 0x98;
```

```
#endif
```

# 常用C51的头文件

```
/*-----
```

```
MATH.H
```

Prototypes for mathematic functions.

Copyright (c) 1988-2002 Keil Elektronik GmbH and Keil Software, Inc.

All rights reserved.

```
-----*/
```

```
#ifndef __MATH_H__
```

```
#define __MATH_H__
```

```
#pragma SAVE
```

```
#pragma REGPARMS
```

```
extern char cabs (char val);
```

```
extern int abs (int val);
```

```
extern long labs (long val);
```

```
extern float fabs (float val);
```

```
extern float sqrt (float val);
```

```
extern float exp (float val);
```

```
extern float log (float val);
```

```
extern float log10 (float val);
```

```
extern float sin (float val);
```

```
extern float cos (float val);
```

```
extern float tan (float val);
```

```
extern float asin (float val);
```

```
extern float acos (float val);
```

```
extern float atan (float val);
```

```
extern float sinh (float val);
```

```
extern float cosh (float val);
```

```
extern float tanh (float val);
```

```
extern float atan2 (float y, float x);
```

```
extern float ceil (float val);
```

```
extern float floor (float val);
```

```
extern float modf (float val, float *n);
```

```
extern float fmod (float x, float y);
```

```
extern float pow (float x, float y);
```

```
#pragma RESTORE
```

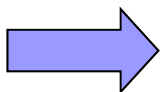
```
#endif
```



# C51 流程控制

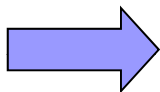
## 1. 选择语句 if

if (表达式)  
{ 语句; }



例: if (p1 != 0)  
{ c=30; }

if (条件表达式)  
{语句1;}  
else  
{语句2;}



例: if (a==b)  
{a++;}  
else  
{a--;}

当 a 等于 b 时,  
a=a+1,  
否则 a=a-1

```
if (表达式1)  
{语句1;}  
else if (表达式2)  
{语句2;}  
else if (表达式3)  
{语句3;}  
... ..  
else if (表达式m)  
{语句m;}  
else  
{语句n;}
```

注：语句中为单一语句，  
可以不用花括弧。

## 2. switch/case语句

switch (表达式)

```
{  
    case 常量表达式1:{语句1;}break;  
    case 常量表达式2:{语句2;}break;  
    case 常量表达式3:{语句3;}break;  
    .....  
    case 常量表达式n:{语句n;}break;  
    default:{语句n+1;}  
}
```

例:

```
switch (k)  
{  
    case 0: {x=1;}      break;  
    case 2: {c=6; b=5;} break;  
    case 3: {x=12;}     break;  
    default: break;  
}
```

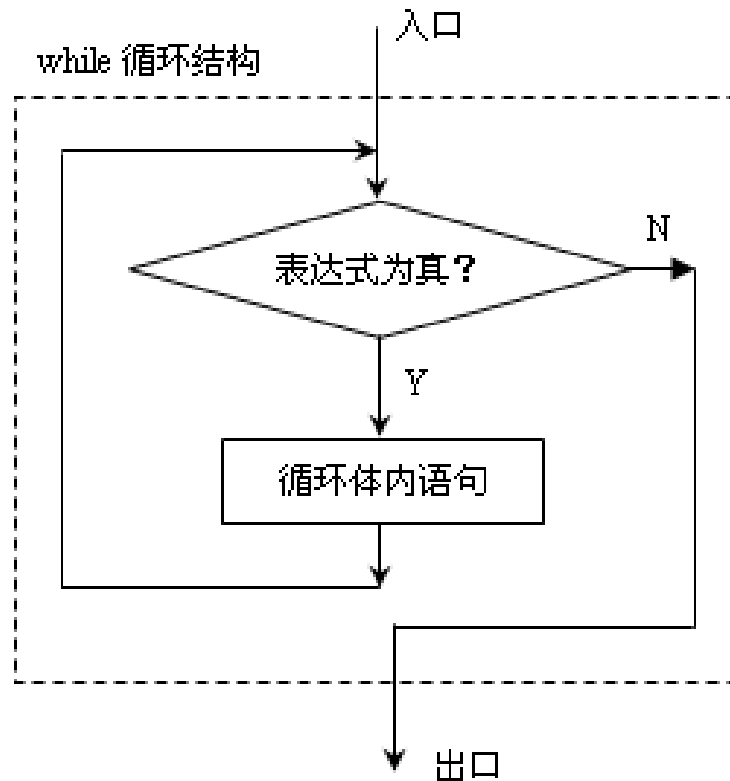
### 3. while语句

```
while (条件表达式真)
{
    语句;
}
```

先判断  
后循环

例

```
while(P0!=0)
{
    x=P0;
}
```

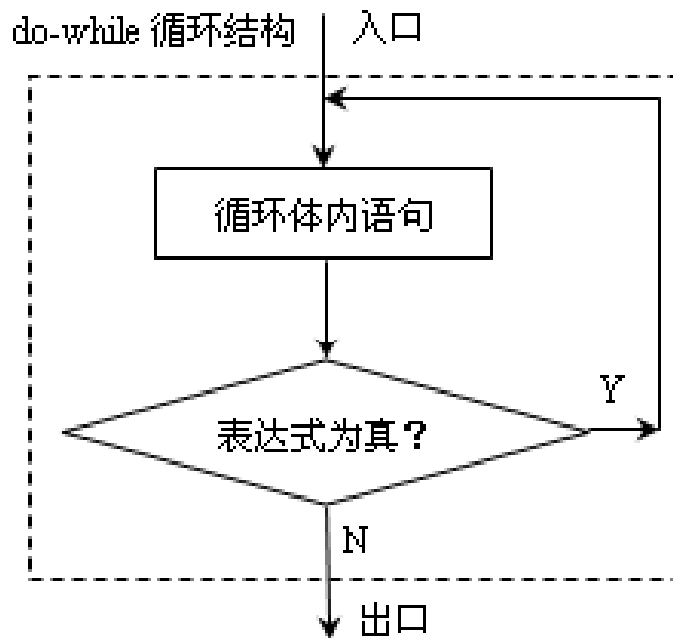


```
do
{
    语句;
}
while (条件表达式真);
```

先循环  
后判断

例

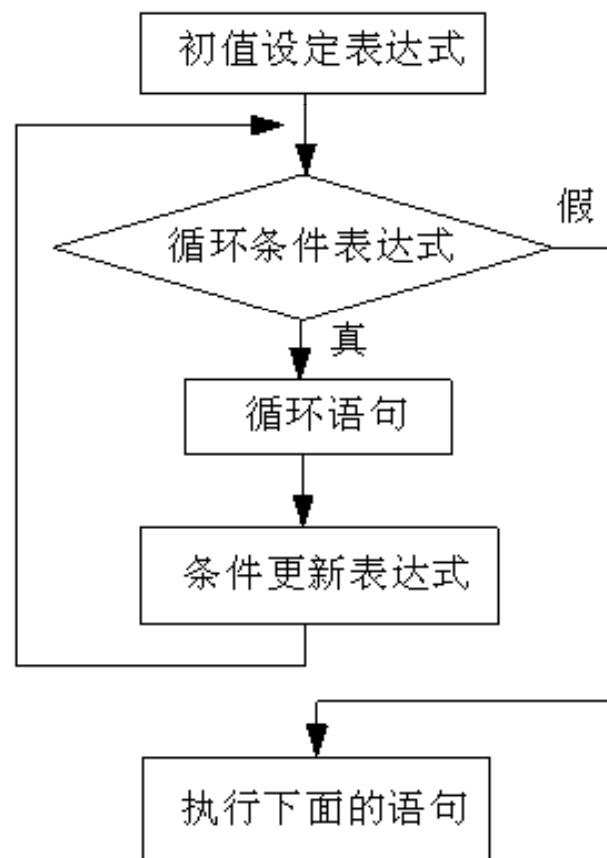
```
int sum=0, i;
do
{
    sum= sum+i ;
    i++;
} while (i<=10)
```



## 4. for语句

```
for ([初值设定表达式]; [循环条件表达式]; [条件更新表达式])  
{  
    循环语句;  
}
```

例 : `int i, sum=0;`  
`for(i=0; i<=10; i++)`  
`{`  
 `sum=sum+i;`  
`}`



## § 4-5 C51程序设计实例

【例1】清零程序(将2000H—20FFH的内容清零)

### ★ 汇编语言程序

```
                ORG 0100H
SE01:  MOV R0, #00H
        MOV DPTR, #2000H      ; (2000H) 送DPTR
L001:   CLR A
        MOVX @DPTR, A         ; 0送 (DPTR)
        INC DPTR              ; DPTR+1
        INC R0                ; 字节数加1
        CJNE R0, #00H, L001   ; 不到FF个字节
LOOP:   SJMP LOOP
```

## 【例1】清零程序(将2000H—20FFH的内容清零)

### ★ C51程序

```
#include <reg51.h>
main( )
{
    int    i;
    unsigned char xdata *p=0x2000;
                                   /*指针指向2000H单元*/
    for(i=0;i<256;i++)
        { *p=0; p++; }          /*清零2000H-20FFH单元*/
}
```

**【例2】查找零的个数**（在2000H--200FH中查出有几个字节是零，把个数放在2100H单元中）

★ 汇编语言程序

```
ORG 0100H
L00: MOV R0, #10H      ;查找16个字节
    MOV R1, #00H
    MOV DPTR, #2000H
L11: MOVX A, @DPTR
    CJNE A, #00H, L16  ;取出内容与00H相等吗?
    INC R1             ;取出个数加1
L16: INC DPTR
    DJNZ R0, L11       ;未完继续
    MOV DPTR, #2100H
    MOV A, R1
    MOVX @DPTR, A      ;相同数个数送2100H
L1E: SJMP L1E
```



**【例2】** 查找零的个数（在2000H--200FH中查出有几个字节是零，把个数放在2100H单元中）

## ★ C51程序

```
#include <reg51.h>
main ( )
{
    unsigned char xdata *p=0x2000;
    int n=0, i;
    for(i=0; i<16; i++)
    {
        if(*p==0) n++;
        p++;
    }
    p=0x2100; *p=n;
}
```

**【例3】** 用**C**程序中插入汇编程序延时的方法，实现从**P1.1**输出周期**10ms**的方波（设时钟频率为**12MHz**）

```
sbit P1_1=P1^1;
void main( )
{
    while(1) {
        P1_1=1;
        #pragma asm
            MOV    R7, #10                ; 1T
        DEL: MOV   R6, #250              ; 1T
            DJNZ   R6, $                  ; 2T
            DJNZ   R7, DEL                ; 2T
        #pragma endasm
        P1_1=0;
    }
}
```

**C51**与汇编语言混合编程  
(生成**SRC**文件)

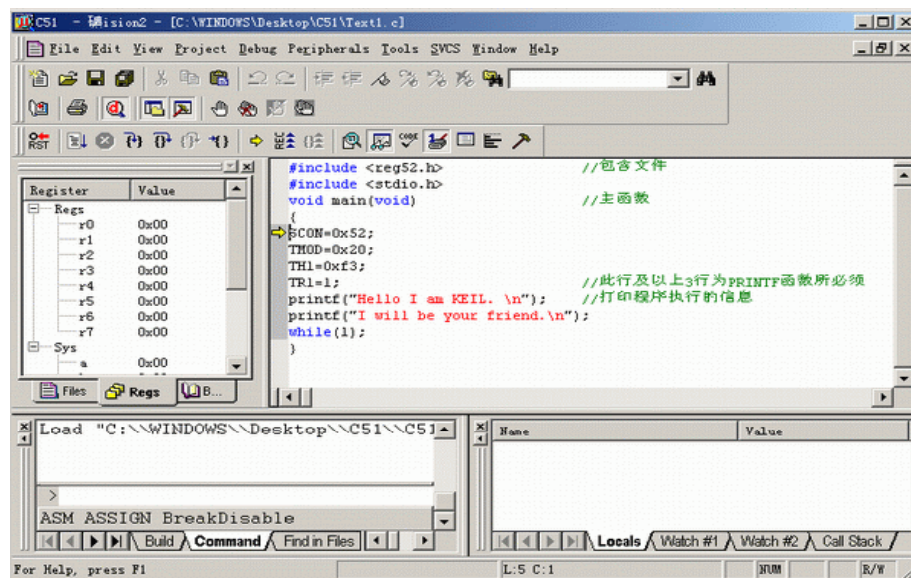
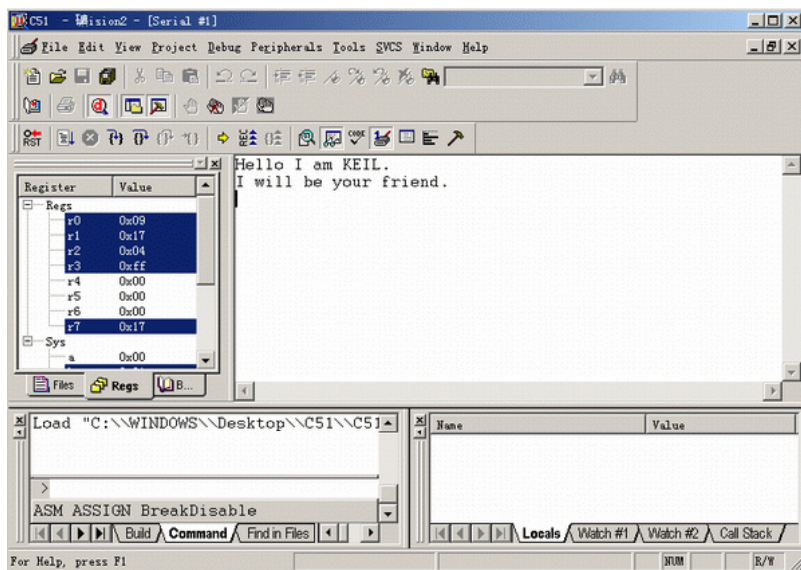
# KEIL C51单片机编程软件

KEIL C51是德国KEIL公司出品的单片机编程软件，用于8051系列单片机。

单片机编程

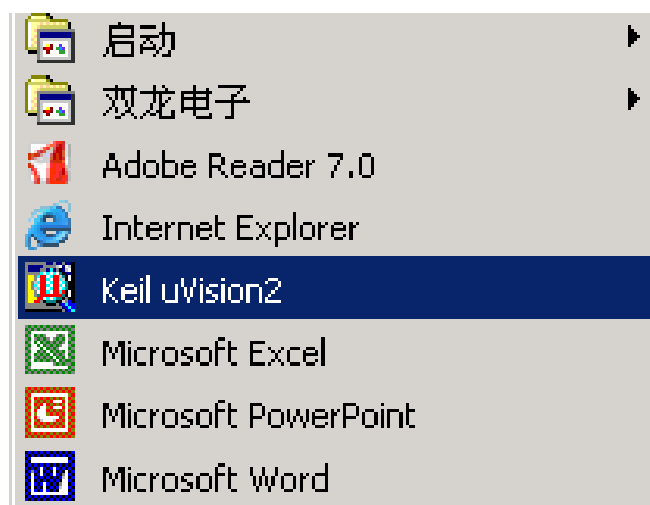
功能

单片机调试：模拟单片机程序运行，观察运行效果，及时发现错误，提高成功率。



# 1、KEIL C51 启动运行:

方法一：“开始” → “程序” → “KEIL uVision2”



方法二：双击“桌面”上的“KEIL uVision2”图标



# KEIL C51 运行界面

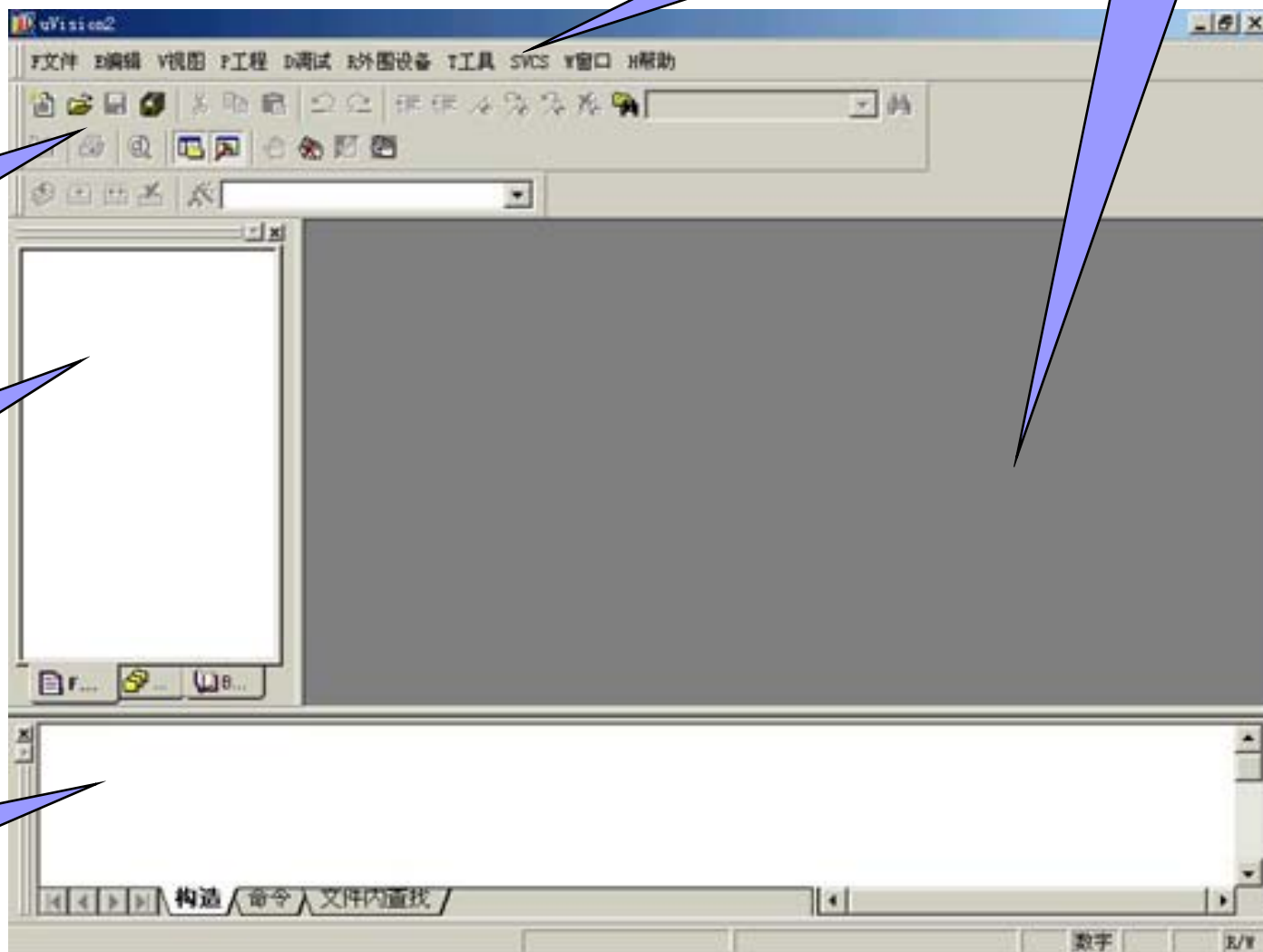
菜单栏

工作区

工具栏

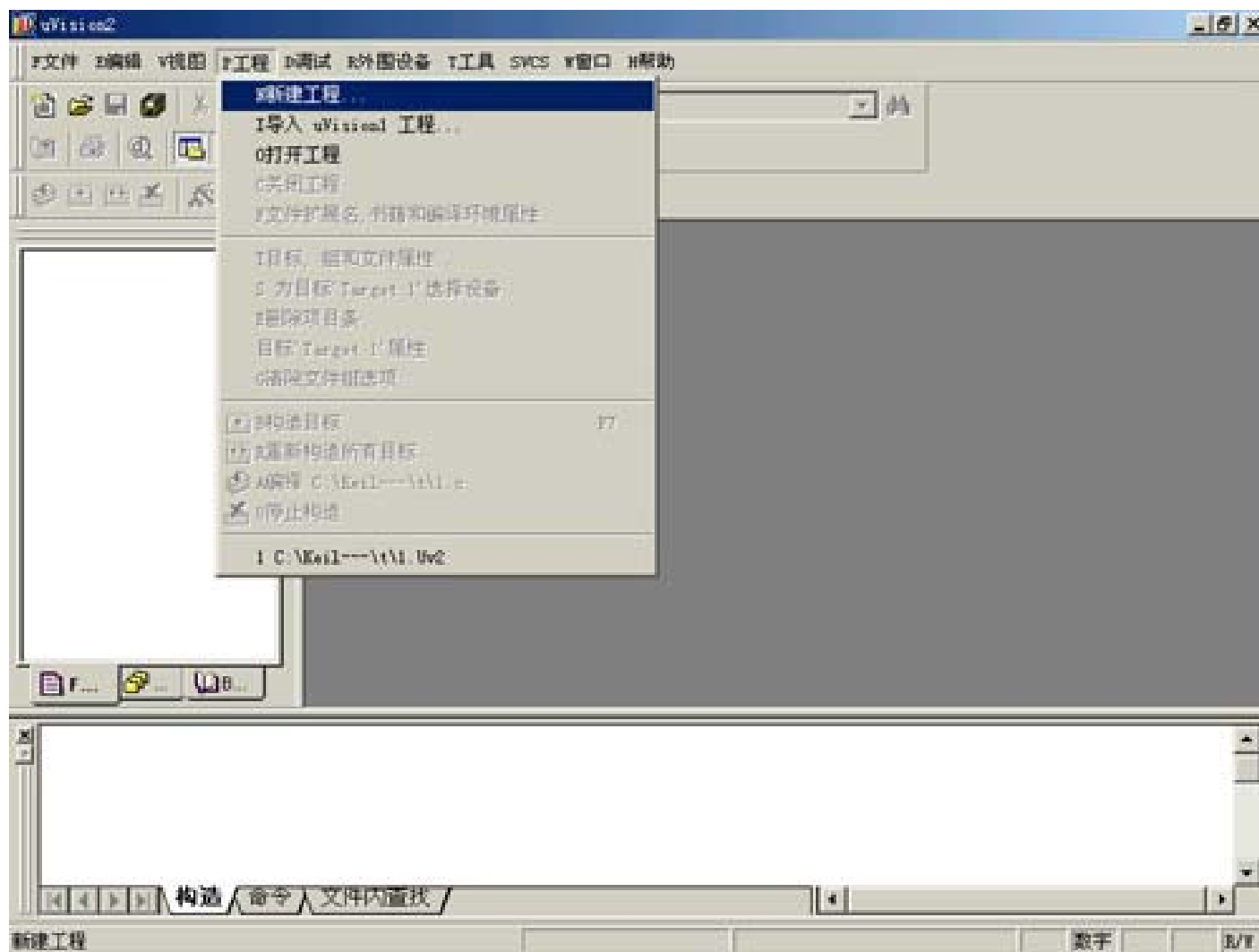
工程窗口

输出窗口

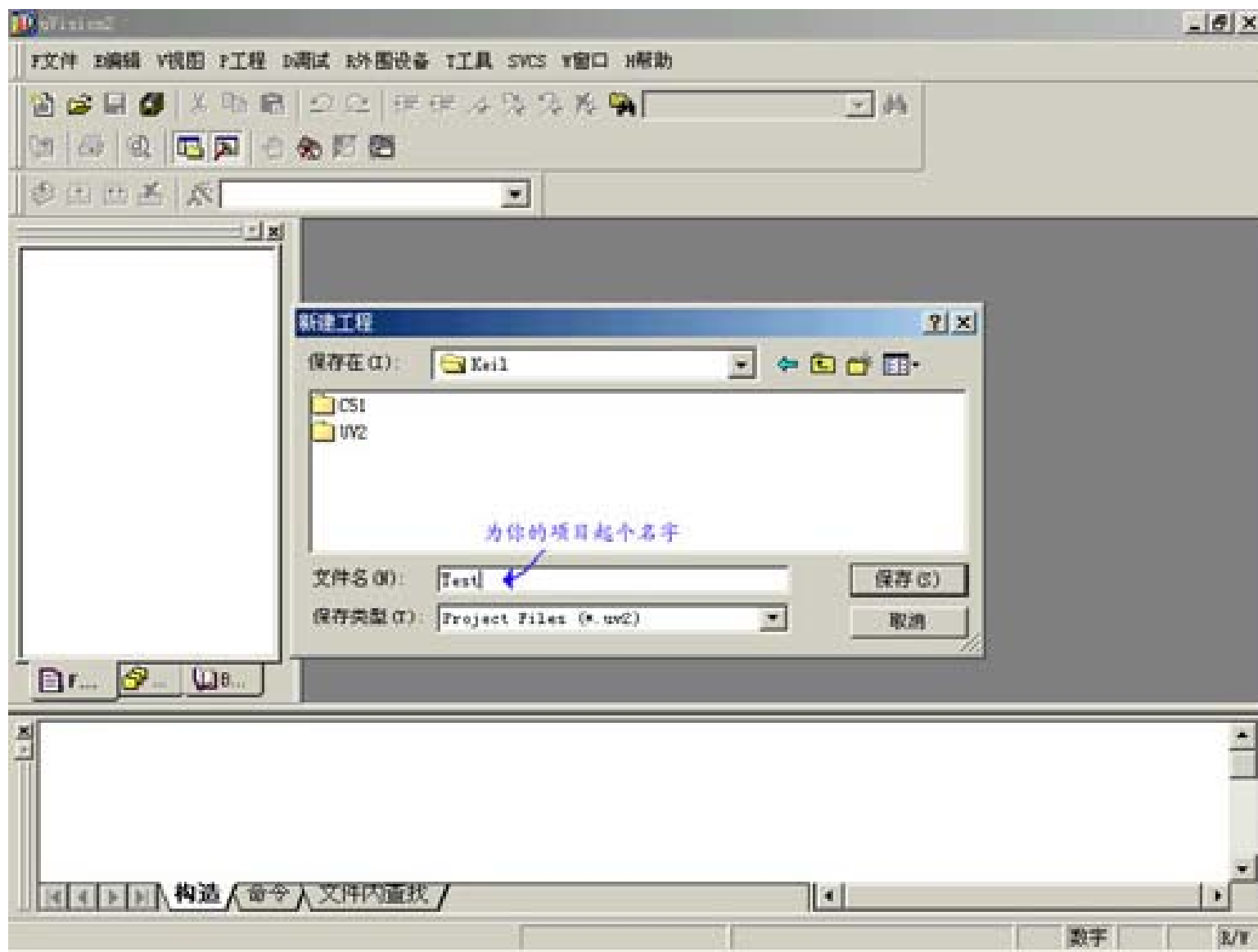


## 2、创建工程并保存：工程用于文件管理

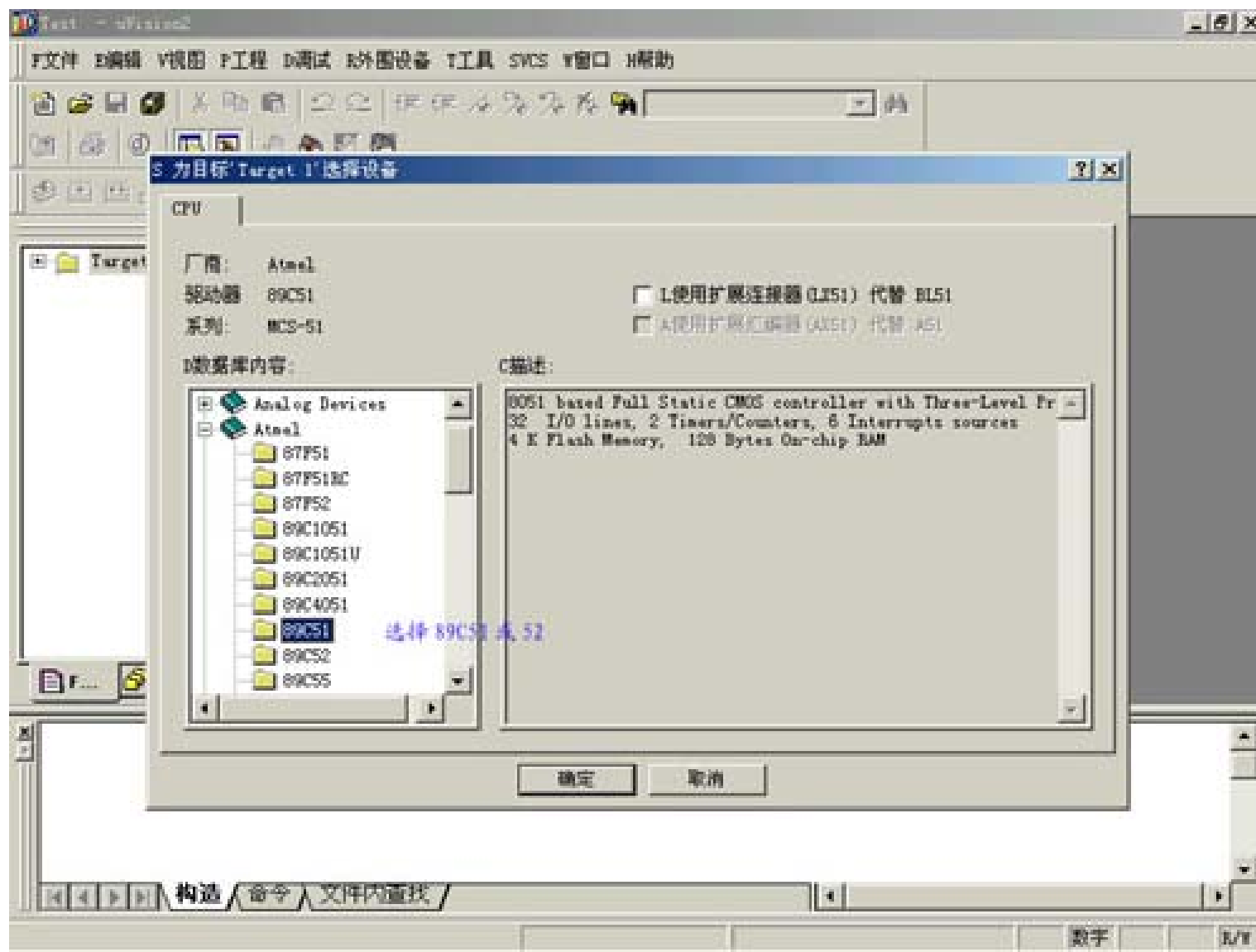
选择：“工程”——“新建工程”



弹出“工程保存”对话框：指定工程名称和保存位置



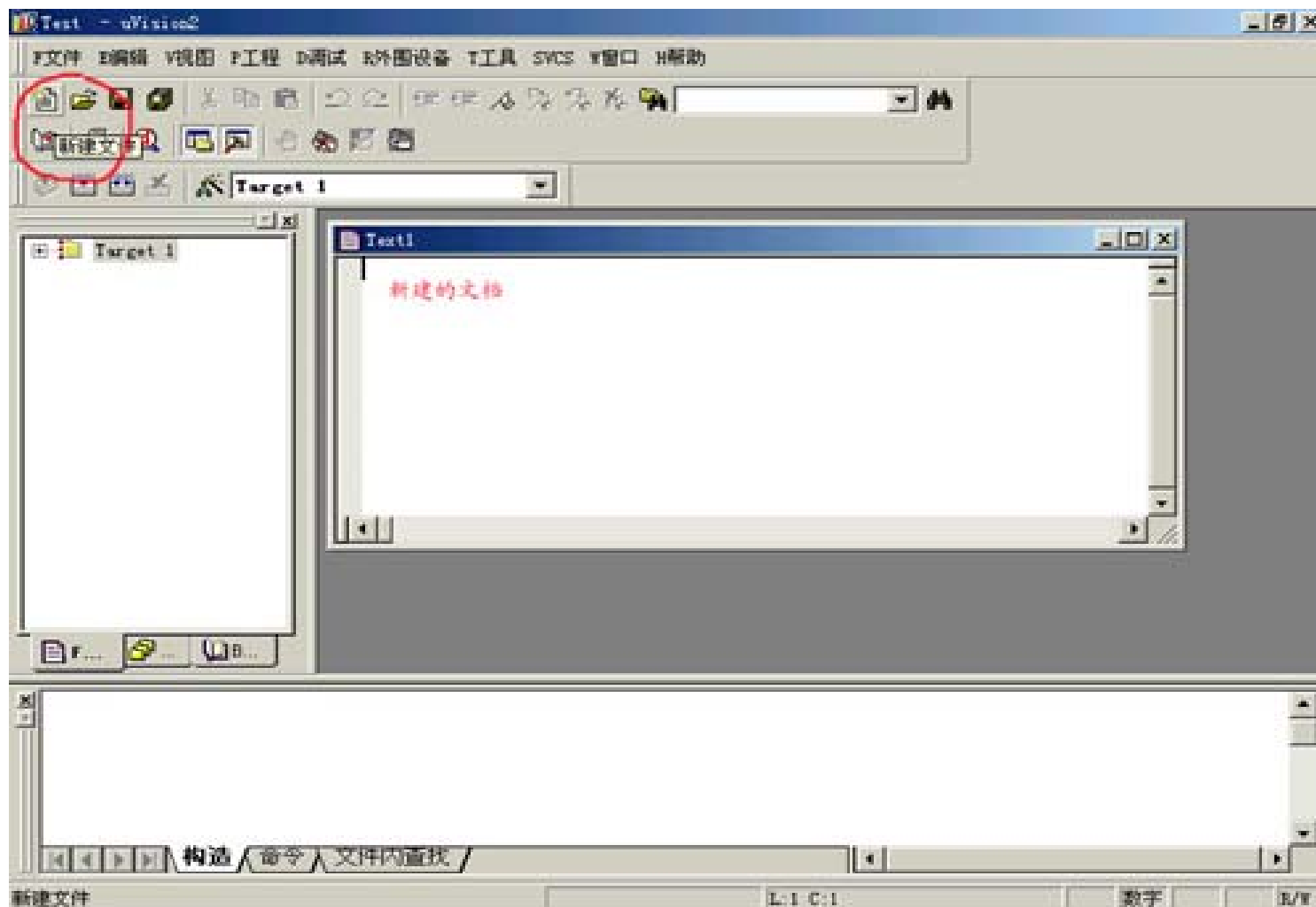
接着弹出“单片机型号”对话框：指定单片机型号



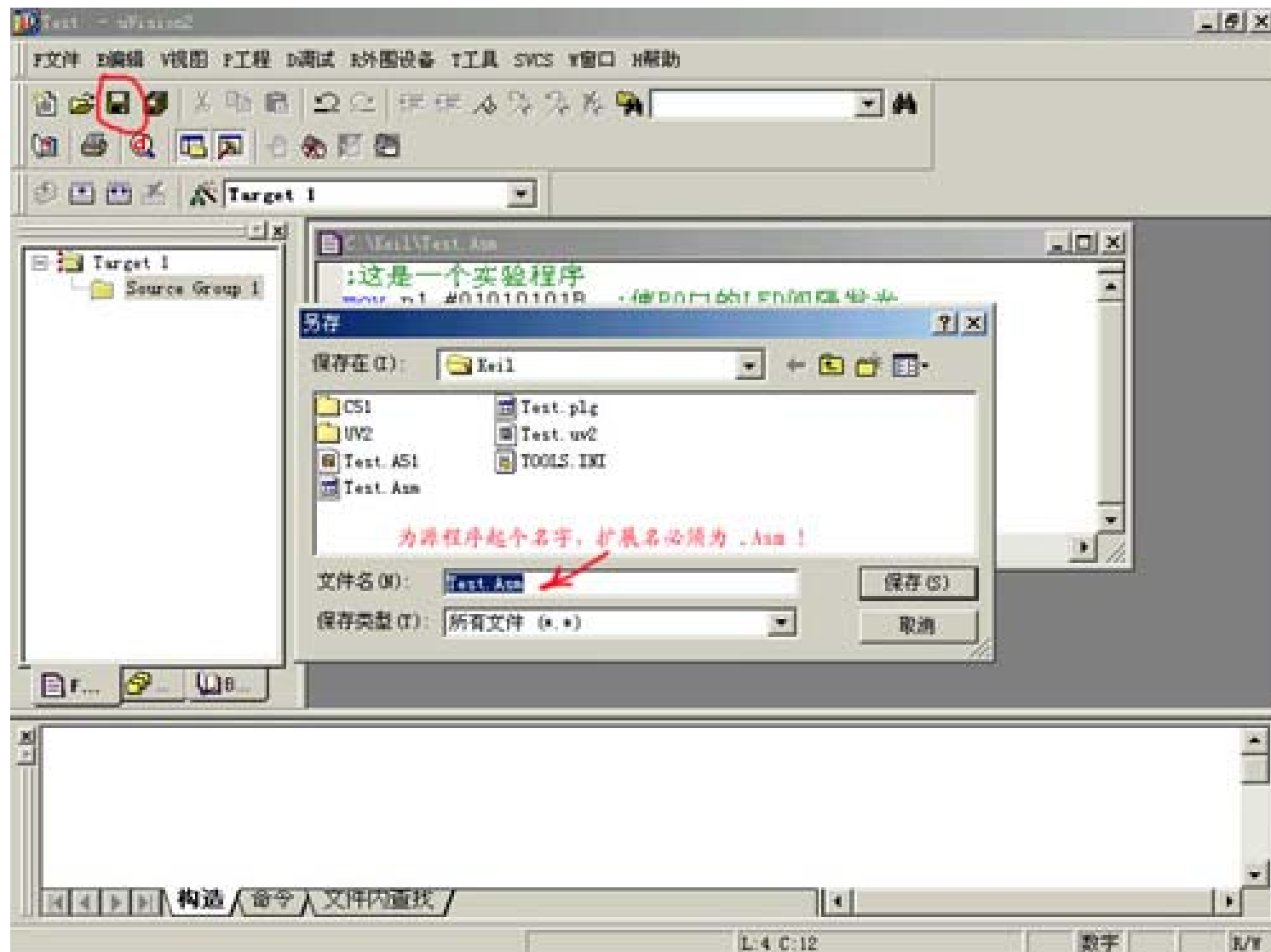


### 3、新建空白文件： 用于编写程序

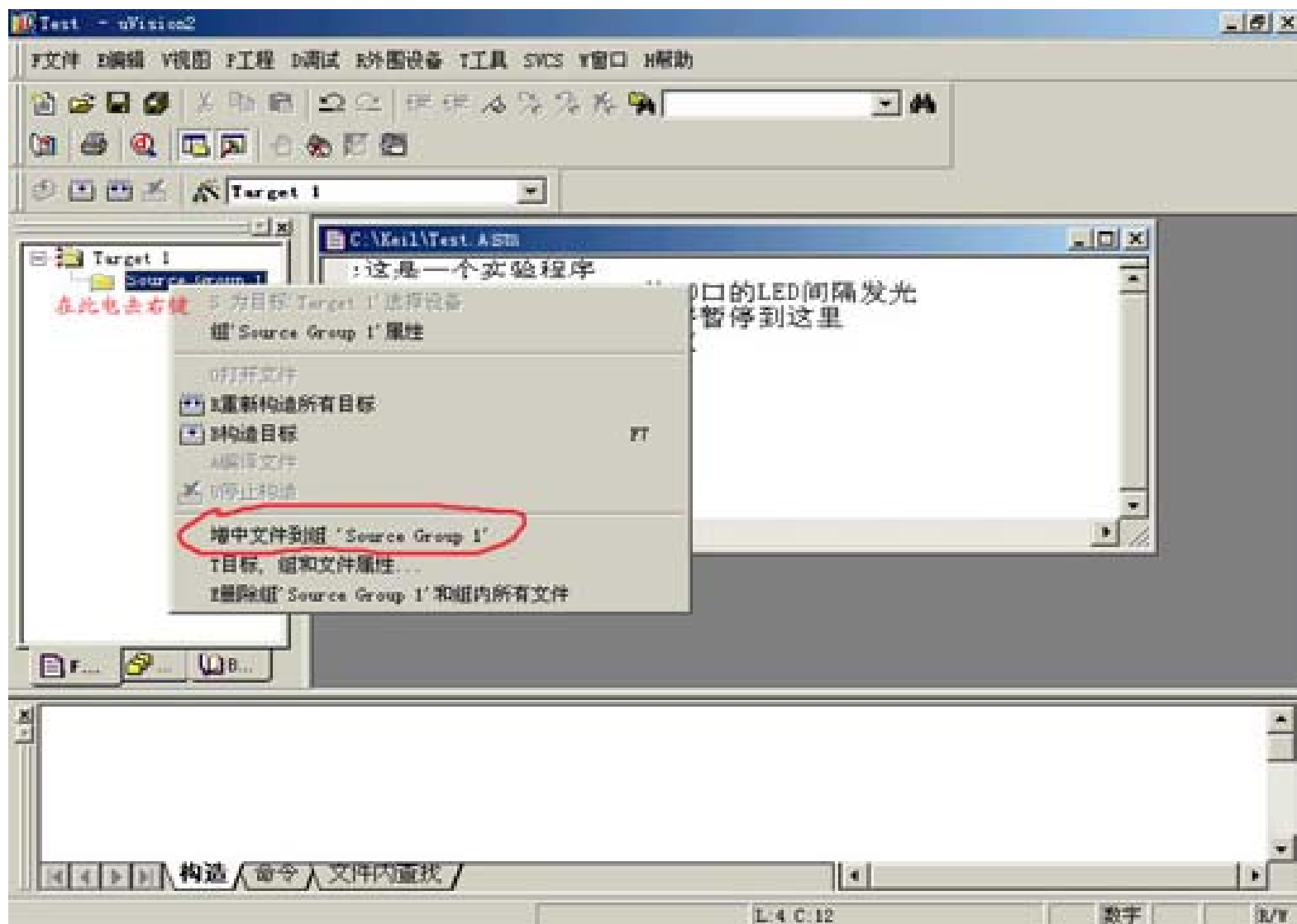
选择：“文件”菜单 → “新建文件”

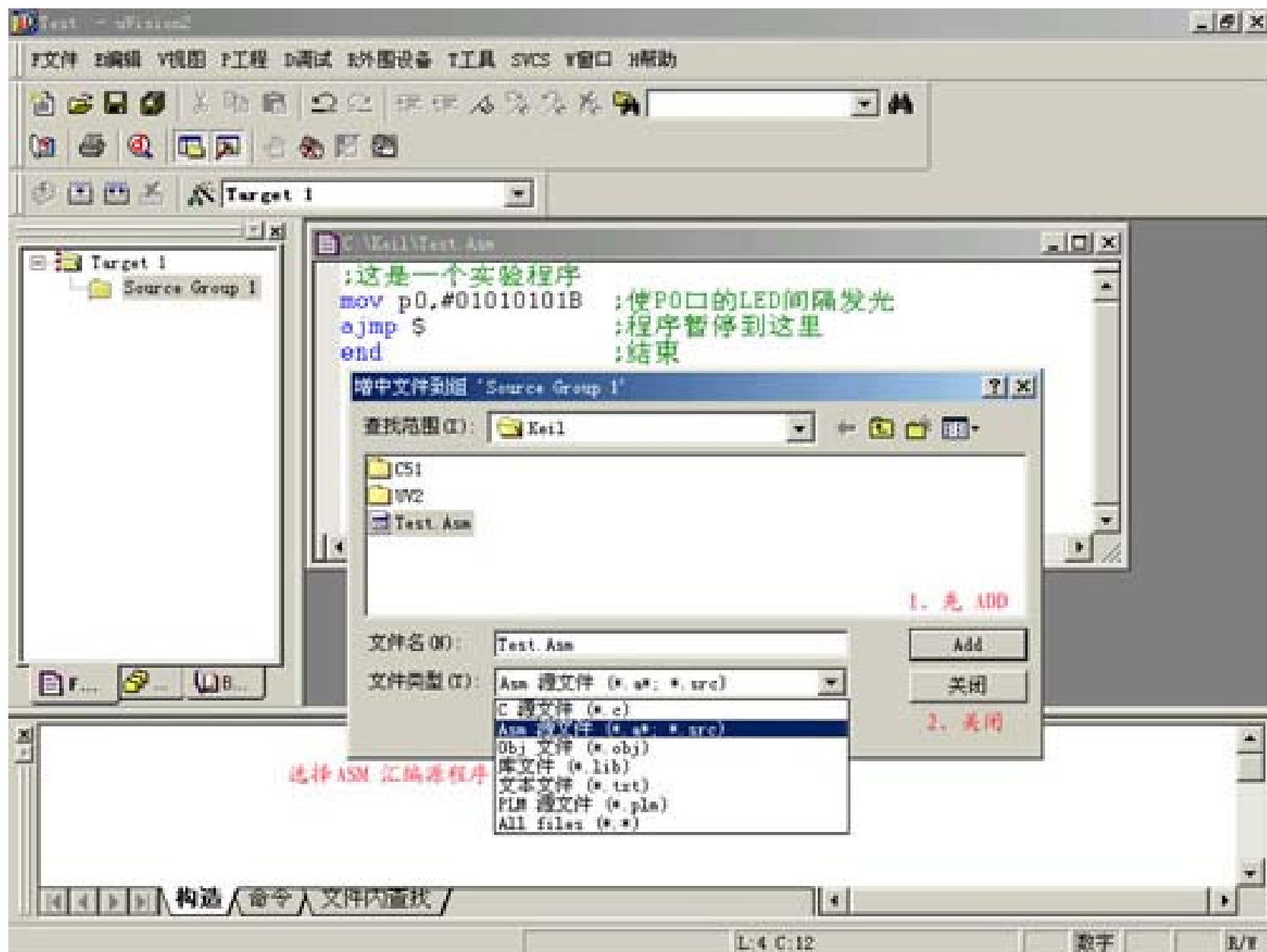


## 4、文件保存： ASM扩展名文件



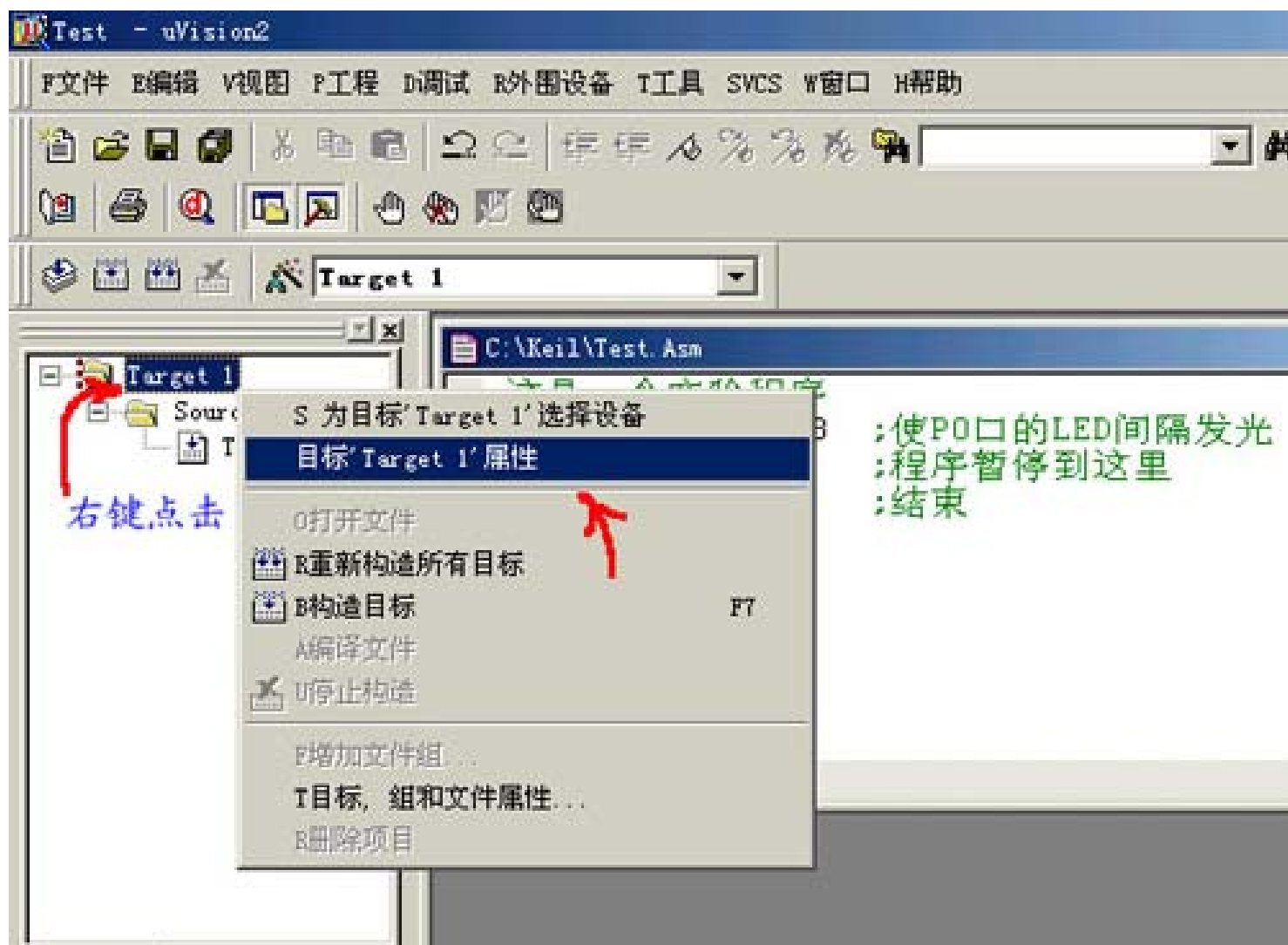
## 5、文件加入工程中：进行编程



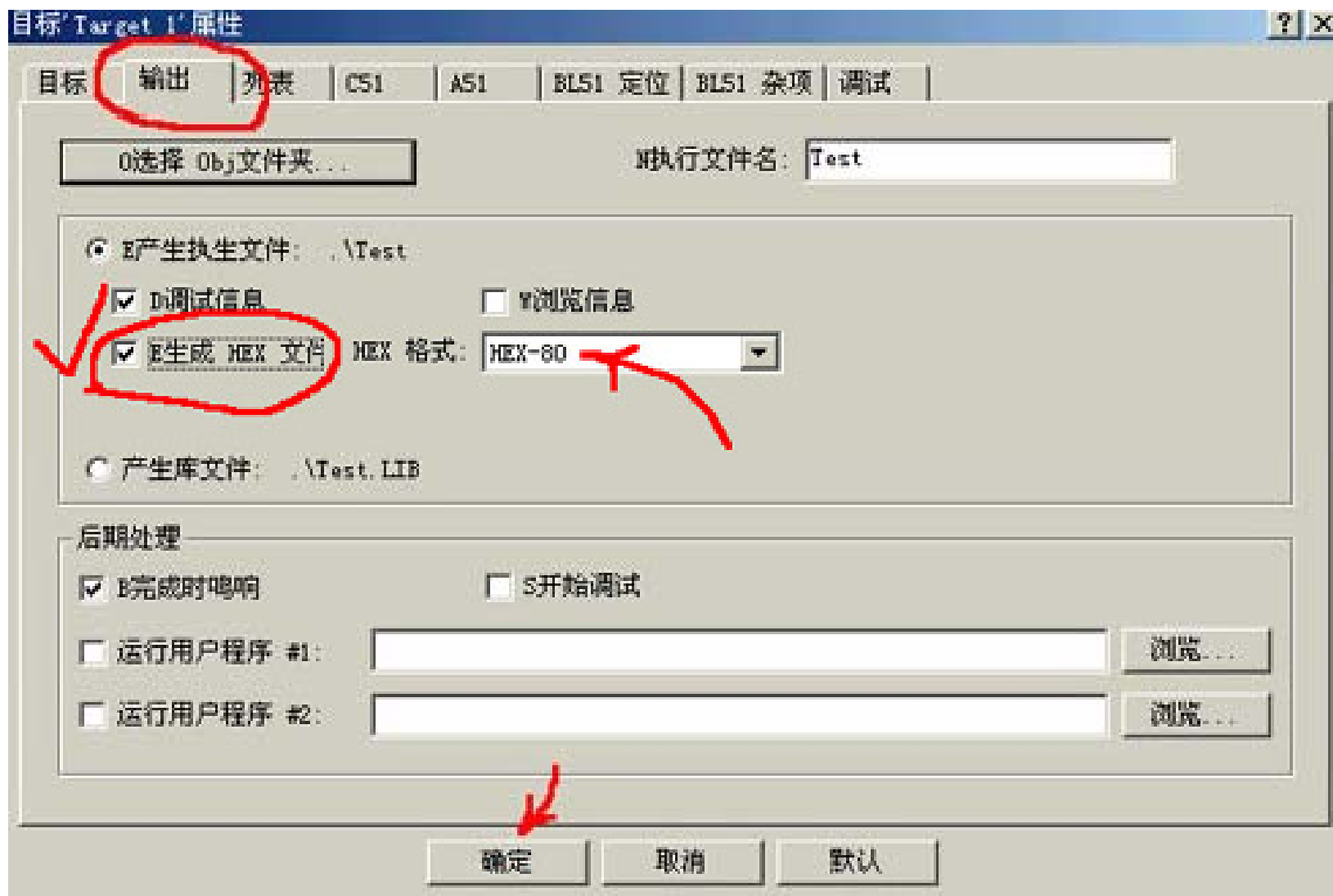


## 6、编译： 用户编写ASM文件 → 单片机HEX执行文件

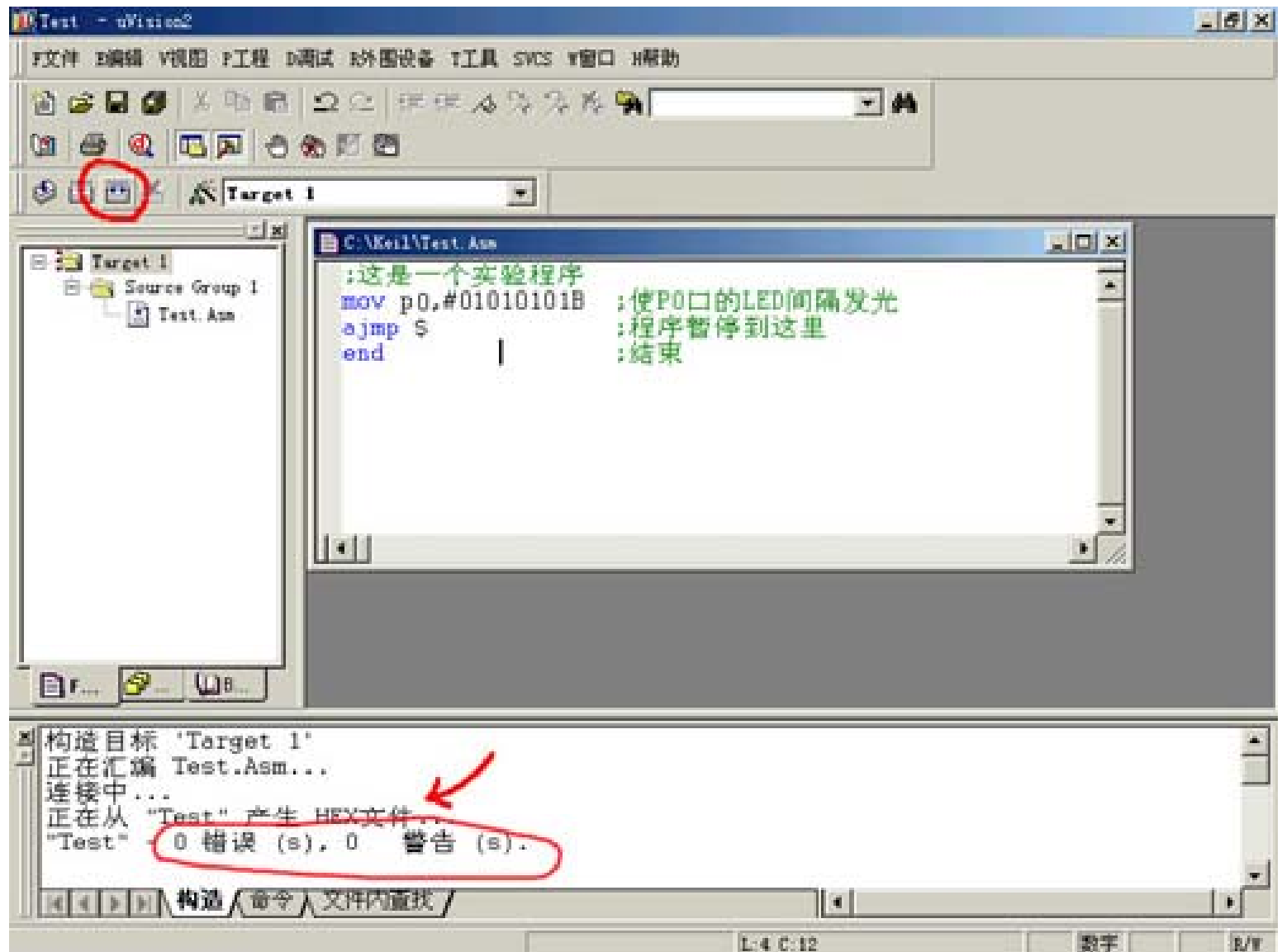
设置 “Target”属性



选择“输出”选项，“E生成HEX文件”选项前要打勾



执行“编译”，产生HEX文件



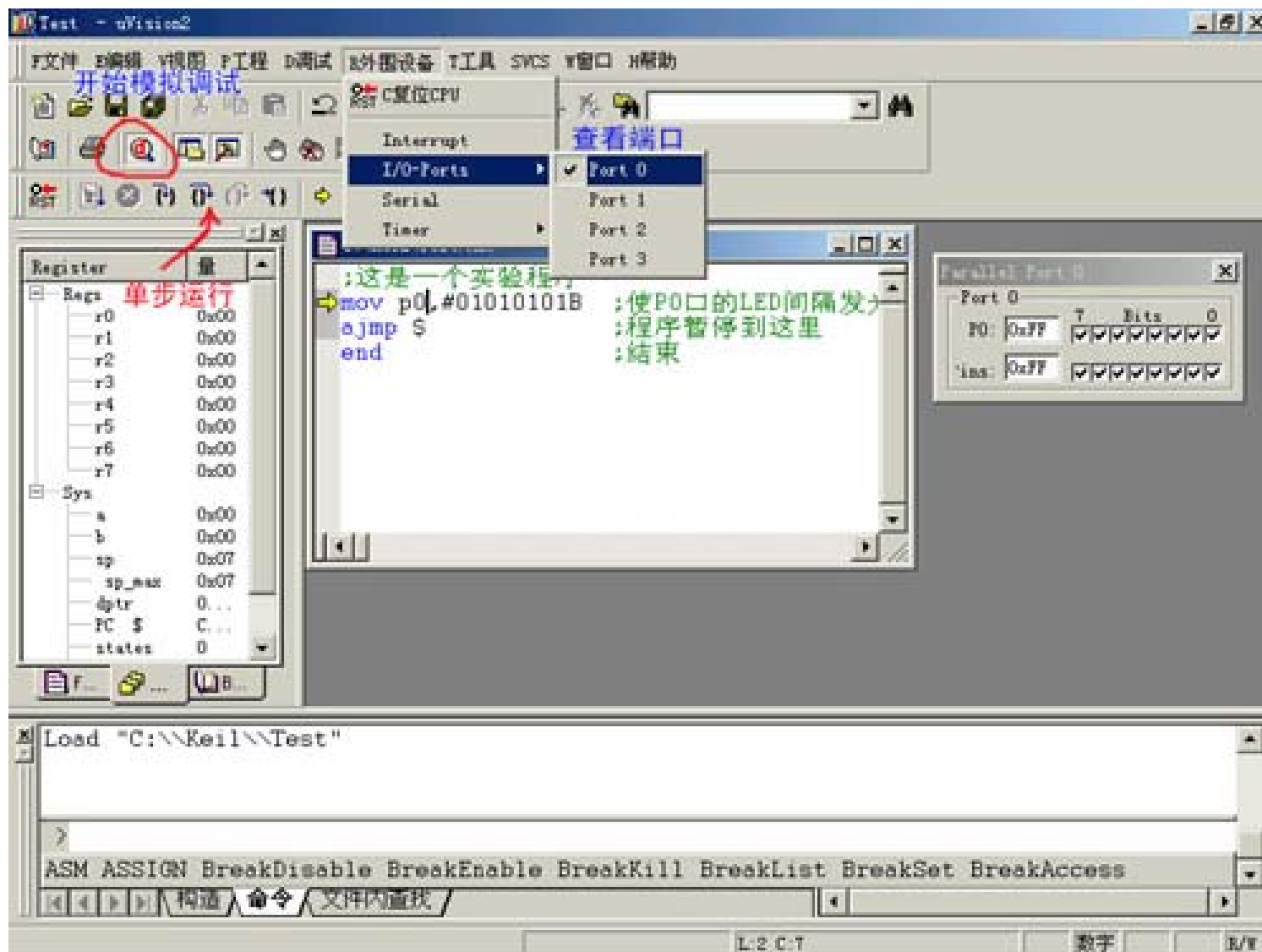
## 7、调试修改：使用**KEIL**软件模拟程序执行

选中“开始/停止调试”项

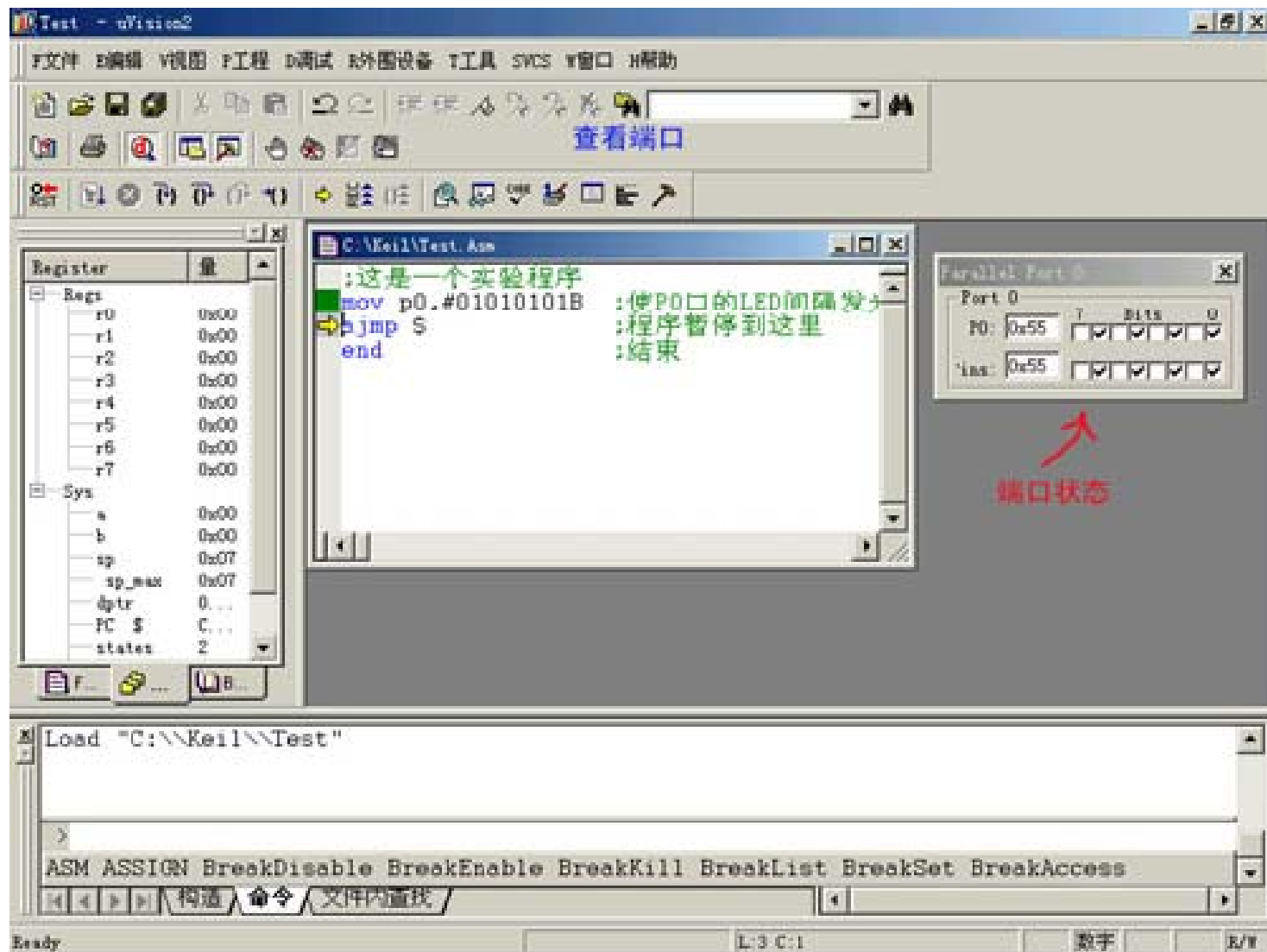




## 进入KEIL调试界面



## 程序模拟运行效果



# 调试界面

The screenshot displays a microcontroller development environment with the following components:

- Project Workspace:** A tree view on the left showing the project structure. The 'Registers' window is highlighted with a red circle and labeled '寄存器区' (Register Area).
- Assembly Code:** The main window shows assembly code for a delay routine. The code is as follows:

```
07 SJMP $
08
09
10
11 DELAY: MOV R5, #100
12 DELO: MOV R6, #200
13 DEL1: MOV R7, #248
14 NOP
15 DEL2: DJNZ R7, DEL2
16 DJNZ R6, DEL1
17 DJNZ R5, DELO
18 RET
19
20
21
22 END
```
- Registers Window:** A table showing the current values of registers. The 'sp' register is highlighted with a red circle and labeled '内部RAM' (Internal RAM).
- Memory Window:** A window showing the contents of memory. The address 'd:0000' is highlighted with a red circle and labeled '内部RAM' (Internal RAM).
- Output Window:** A window at the bottom left showing the command prompt output. The text 'ASM ASSIGN BreakDisable BreakEnable BreakKill BreakList BreakSet' is visible.
- Simulation Status:** The bottom status bar shows 'Simulation' and 't1: 0.00000500 sec'.

Additional annotations include a red circle around the 'Registers' window, a red circle around the 'Memory' window, and a red circle around the 'Simulation' status bar.

# KEIL C51 使用步骤:

